SE 350 - Operating System
Project - Part 1 Documentation

Andrew Gillies
Delia Yao
Joohee Lee
Yiying Ho
Due date: Jan 30, 2014

---

1. System Processes

1.1 Purpose

The system processes are needed by system to perform a basic service in operating system. The user processes what were implemented in part one of the project were to operate at unprivileged level in order to demonstrate the system's operation. Therefore, the main purpose of having multiple test process is to determine if the system performs expectedly.

1.2 User Processes

The system processes that implemented in part one were one null process and six user test processes. Those processes were able to show that if the system is running correctly. Since, the null process has the lowest priority than all of the test processes, hence null process will not run until all the test processes are in the block queue. The test processes are needed by demonstrate the operation of the system, so they had been assigned to different priority and different ID in order to test the API of the system.

Initializing the processes in `set_test_procs()`, which the processes had been assigned the priority, ID, stack size, and the start pointer of program counter. In this specific senior, the process one and process two had priority high, process three and four has priority priority medium and the process five and six had priority low.

There are four different APIs need to be tested: `request_memory_block()`, `release_memory_block(p_mem_blk)`, `set_process_priority(1, 0)`, and `get_process_priority(1)`. In order to test all the APIs and to make all the test processes run, there were the six test cases to make sure the system is running expectedly and each process had different behaviour in order to indicate the system operation.

1.2.2 Behaviour of Processes

The behaviour of each process was different than each other, so it will be able to call different API of RTX and show if the system is running expectedly.

1.2.2.1 Process One

Since the process one had the highest priority than other process and it was the first one been called, hence the process one was the first process in the ready queue initially.

```
void proc1(void) {
	while(1){
		for(i = 0; i < NUM_BLOCK; i ++) {
		mem_ptr = request_memory_block();
	}
	release_processor();

	release_memory_block(mem_ptr);
	set_process_priority(1, 3);

	release_processor();

	}
}
```

pseudocode 1: The behaviour of process one

As the pseudocode 1 is shown above, the process ran inside the infinite loop, it initial requested all the memory blocks in the for loop and once it got all the memory blocks, it released the processor, which the operating system gained the control. Once the process one gained the control again, it would start releasing one memory block and then change the priority to the lowest before operating system gained the control.

1.2.2.2 Process Two

Since the process two also had the highest priority than other process and it was the second one been called, hence the process one was the second process behind the process one in the ready queue initially.

```
void proc2(void){
void* mem_ptr;
while(1){
		mem_ptr = request_memory_block();
		release_processor();
	}
}
```

pseudocode 2: The behaviour of process two

As the pseudocode 2 is shown above, the process ran inside the infinite loop and it requested one memory block every time when it gained control from operating system.

1.2.2.3 Process Three

Since the process three has medium priority which means, it will only run when all high priority processes are in block queue. Also, because process three is the first medium priority process been called, so it will be the first medium process gain the control from operating system when all high priority processes are blocked.

```
void proc3(void){
int array_size = NUM_BLOCK - 5;
      while(1){
            for(i = 0; i < array_size; i++){
                  mem_ptr = request_memory_block();
            }
      release_processor();

      release_memory_block(mem_ptr);

      release_processor();
      }
}
```

pseudocode 3: The behaviour of process three

As the pseudocode 3 is shown above, the process ran inside the infinite loop. It requested 5 memory blocks less than the total memory blocks and then gave the control back to operating system. Once it gained the control again, it will release one memory block. After releasing the memory block, it gave control back to operating system.

1.2.2.4 Process Four

Since the process four has medium priority so it will only have chance to run when all high priority processes are in block queue. In addition, this process is the second medium process had been called, so it will be running after the process three.

```
void proc4(void){
```

```
	while (1) {
		while(i < 1000){
		i++;
	}
	release_processor();

	mem_ptr = request_memory_block();
	release_processor();
	}
}
```

pseudocode 4: The behaviour of process four

As the pseudocode 4 is shown above, the process ran inside the infinite loop. When first time it gained the control, the process would not do anything but counting from 0 to 1000. Then, it released the processor after it counted until 1000. After, when operating system gave the control to the process 4 again, it request one memory blocks and then gave the control back to operating system.

1.2.2.4 Process Five

Since the process five has low priority so it will only have chance to run when all high priority processes and medium processes are in block queue. Since, it was the first low priority process had been called, therefore, it was the first low priority process to run in the ready queue.

```
void proc5(void){
	while(1) {
		for(i = 0; i < 10; i ++) {
			mem_ptr[i] = request_memory_block();
		}
	release_processor();
	for(i = 0; i < 10; i ++) {
		release_memory_block(mem_ptr[i]);
	}
	release_processor();
	}
}
```

pseudocode 5: The behaviour of process five

As the pseudocode 5 is shown above, the process ran inside the infinite loop, and it requested 10 memory blocks when first time it ran and then release the processor. After it gained the control back, it released all 10 memory blocks.

1.2.2.4 Process Six

The process six is the second low priority process in the ready queue. So it would not have chance to run until all the processes were blocked.

```
void proc6(void){
      release_processor();
}
```

pseudocode 6: The behaviour of process six

As the pseudocode 6 is shown above, the process six did not request or release any memory blocked so it would keep running. The process six would print the test result when the test ended.

1.2.3 Test Cases and Expectation

The test cases helped the RTX determine the correctness of operating system. Therefore, there are six different test cases to check if APIs of RTX working as expected.

The first test case is to test if the process tried to request a memory block but got blocked where there is no more memory block available. The process one requested all memory blocks in the beginning. After release the processor, process two would start running and try to request one memory block as well. If the operating system is working correctly, process two should be blocked and put it into the block queue. In order to show the correctness of the system, the process two should get blocked before it finished requesting the memory. Therefore, there was a test flag for test one, `test[1]`, it indicated that if process two kept running after it requested the memory block, then the flag would be setted to `-1000`, which means that it failed the first test case.

The second test case is to test the process should be unblocked as long as there are enough memory blocks to be requested. After first test done, the process one should be ready to release the memory block. The process two should be unblocked immediately after process one release one memory block. In order to test the second test case, the test flag for test two, `test[2]`, it checked that the system supposed to start running after requesting the memory in process two. If it ran before requesting the memory, then the system did not run as expected.

After the process two got the memory block, the process one would get the control and it would set the priority of itself to be the lowest. It would last the control right after it changed the priority. Then in the third test case, it would check if the first process has the lowest priority. Since process one changed the priority of itself, then process two would be the only one in the ready queue. The process two would be blocked once it tried to request the memory again but there was no available memory block, since process one originally had all the memory blocks

and only release one block before it changed to the lowest priority. Therefore, process three should be start running as expected. Beginning of the process three, it checked if the priority of process one is the lowest. After testing the priority of the process one, it would also test if process three got the chance to run. In order to test that, there is a test flag for test four, `test[2]`, which it checked that if the system gave the control back to process one after it changed the priority of itself, then the flag would be setted to `-1000`, otherwise, it would not be `-1000.`

The test case five and six were both trying to show that if any of process tried to request memory blocked, then it would be blocked into the block queue. After all the testing, the process six would keep running because it did not request or release any memory block, hence, it would not be blocked and it would be running consistently.

1.3 Results

After running all the test result, the APIs of RTX pass all the test cases which means, the operating system runs correctly and expectedly.

2. Lessons Learned

The code that was supplied for part one was enough to guide us in the right direction for our operating system. Most of our problems came from bugs in our program rather than conceptual errors. We had a large problem with pointer arithmetic, which resulting in allocating a lot more memory than intended on our heap. Eventually we realized how C handled pointer arithmetic and were able to solve our problem. We also faced an issue where our program would hard fault and we couldn't find out why. After much debugging we realized that our process was just putting too many variables on the stack and was resulting in stack overflow. Luckily the debugger is actually pretty good and made it doable for us to find these problems.

At first we had a lot of trouble recognizing how the request memory function actually worked (based on the tutorial slide pseudo code). It took us awhile to realize how the OS would switch context while waiting for a memory block and then return to that same place once a block became free. Overall it was a long hard assignment but very rewarding!